

```

// NDDL Grammar (Without semantic checks)

class NddlParser extends Parser;

nddl: (nddlStatement)*
    ;

nddlStatement:
    inclusion
    | enumeration
    | constraintSignature
    | typeDefinition ";"
    | variableDeclaration ";"
    | classDeclaration
    | rule
    | allocation ";"
    | assignment ";"
    | function ";"
    | constraintInstantiation ";"
    | relation ";"
    | goal ";"
    | ";"
;

inclusion: "#include" STRING
    ;

enumeration: "enum" IDENT symbolSet
    ;

symbolSet: "{" (symbolDefinitions)? "}"
    ;

symbolDefinitions: symbolDefinition ("," symbolDefinition)*
    ;

symbolDefinition: IDENT
    ;

typeDefinition: "typedef" typeWithBaseDomain IDENT
    ;

typeWithBaseDomain: "int"      (intervalIntDomain|enumeratedIntDomain)?
    | "float"    (intervalFloatDomain|enumeratedFloatDomain)?
    | "boolean"  (enumeratedBoolDomain)?
    | "string"   (enumeratedStringDomain)?
    | IDENT      (enumeratedSymbolDomain|enumeratedObjectDomain)?
    ;

constraintSignature: "constraint" IDENT typeArgumentList
    ("extends" IDENT typeArgumentList)?
    (signatureBlock | ";" )
    ;

signatureBlock: "{" (signatureExpression)? "}"
    ;

signatureExpression: signatureAtom ("&&" | "||") signatureAtom)*
    ;

signatureAtom: "(" signatureExpression ")"

```

```

        | IDENT "<:" (type | "numeric" | IDENT )
        ;

classDeclaration: "class" IDENT (
        ("extends" IDENT)? classBlock
        | ";" )
        ;

classBlock: "{" (classStatement)* "}"
        ;

classStatement: constructor | predicate | variableDeclaration ";" | ";"
        ;

constructor: IDENT constructorParameterList constructorBlock
        ;

constructorBlock: "{" (constructorStatement)* "}"
        ;

constructorStatement: (assignment | superInvocation) ";" | ";"
        ;

constructorParameterList: "(" (constructorParameters)? ")"
        ;

constructorParameters: constructorParameter ("," constructorParameters)?
        ;

constructorParameter: type IDENT
        ;

predicate: "predicate" IDENT predicateBlock
        ;

predicateBlock: "{" (predicateStatement)* "}"
        ;

// Note: Allocations are not legal here.
predicateStatement: ( variableDeclaration | constraintInstantiation | assignment)? ";"
        ;

rule: IDENT "::<" IDENT ruleBlock
        ;

ruleBlock: "{" (ruleStatement)* "}" | ruleStatement
        ;

ruleStatement: (relation | variableDeclaration | constraintInstantiation) ";" | flowControl | ";"
        ;

type: "int" | "float" | "boolean" | "string" | IDENT
        ;

relation: (IDENT | "this")? temporalRelation predicateArgumentList
        ;

goal: ("rejectable" | "goal") predicateArgumentList
        ;

predicateArgumentList: IDENT | "(" (predicateArguments)? ")"

```

```

        ;
predicateArguments: predicateArgument ("," predicateArgument)*
        ;
predicateArgument: qualified (IDENT)?
        ;
constraintInstantiation: IDENT variableArgumentList
        ;
constructorInvocation: IDENT variableArgumentList
        ;
superInvocation: "super" variableArgumentList
        ;
variableArgumentList: "(" (variableArguments)? ")"
        ;
variableArguments: variableArgument ("," variableArgument)*
        ;

// Note: Allocation not legal here
variableArgument: anyValue
        ;
typeArgumentList: "(" (typeArguments)? ")"
        ;
typeArguments: typeArgument ("," typeArgument)*
        ;
typeArgument: IDENT
        ;

domain : intLiteral | intervalIntDomain| enumeratedIntDomain | floatLiteral | intervalFloatDomain| enumer
        | enumeratedStringDomain | enumeratedBoolDomain | enumeratedSymbolDomain
        ;
intervalIntDomain: "[" intLiteral (",")? intLiteral "]"
        ;
intervalFloatDomain: "[" floatLiteral (",")? floatLiteral "]"
        ;
enumeratedIntDomain: "{" intSet "}"
        ;
intSet: intLiteral ("," intLiteral)*
        ;
enumeratedFloatDomain: "{" floatSet "}"
        ;
floatSet: floatLiteral ("," floatLiteral)*
        ;
enumeratedObjectDomain: "{" objectSet "}"
        ;

```

```

objectSet: (constructorInvocation|qualified) ("," (constructorInvocation|qualified))*
        ;

enumeratedSymbolDomain: "{" qsymbolSet "}"
        ;

qsymbolSet: qualified ("," qualified)*
        ;

enumeratedStringDomain: "{" stringSet "}"
        ;

stringSet: STRING ("," STRING)*
        ;

enumeratedBoolDomain: "{" boolSet "}"
        ;

boolSet: boolLiteral ("," boolLiteral)*
        ;

flowControl: ("if" expression ruleBlock ("else" ruleBlock )?)
        | ("foreach" "(" IDENT "in" qualified ")" ruleBlock )
        ;

// Note: Allocation not legal here
expression: "(" anyValue ("=="|"!=") anyValue)? ")"
        ;

allocation: "new" constructorInvocation
        ;

variableDeclaration: ("filter")? type nameWithBase ("," nameWithBase)*
        ;

nameWithBase: IDENT "(" anyValue ")" )?
        | IDENT "=" anyValue
        ;

assignment: qualified ("in"|"=") anyValue
        ;

anyValue: STRING | boolLiteral | qualified | domain | allocation
        ;

qualified: "this"
        | ("this.")? IDENT ("." IDENT)*
        ;

temporalRelation: "any" | "ends" | "starts" | "equals" | "equal"
        | "before" | "after" | "contains" | "contained_by"
        | "ends_before" | "ends_after" | "starts_before_end" | "ends_after_start"
        | "contains_start" | "starts_during" | "contains_end" | "ends_during"
        | "meets" | "met_by" | "parallels" | "paralleled_by"
        | "starts_before" | "starts_after"
        ;

intLiteral: INT | "+inf" | "-inf"
        ;

floatLiteral: FLOAT | "+inff" | "-inff"

```

```
        ;

boolLiteral: "true" | "false"
        ;

function : qualifiedName DOT
        ( "specify" variableArgumentList
        | "free" variableArgumentList
        | "constrain" variableArgumentList
        | "merge" variableArgumentList
        | "activate()"
        | "reset()"
        | "reject()"
        | "cancel()")
        | (IDENT ".")? "close()"
        ;

tokenNameList: "(" (tokenNames)? ")"
        ;

tokenNames: IDENT ("," IDENT)*
        ;
```